Summary of the IHFS Data Access Prototype Project

I. Overview of IHFS Data Access Prototype project

The mission statement of the Hydrology Laboratory (formerly the Hydrologic Research Lab) is to "conduct studies, investigations and analyses leading to the application of new scientific and computer technologies for hydrologic forecasting and related water resources problems." In support of that mission, the purpose of the Integrated Hydrologic Forecast System (IHFS) project is to provide a more effective system of tools and techniques for use at River Forecast Centers (RFCs), Weather Forecast Offices (WFOs) and supporting headquarters offices. Initial IHFS efforts laid a foundation of components for evolving the future system. One key component is a Logical Data Model (LDM) (http://hsp.nws.noaa.gov/oh/hrl/ihfs/logmodel/index.html), which provides definitions and identifies relationships for all the data used by hydrologists. The second key ingredient is a software architecture (http://hsp.nws.noaa.gov/oh/hrl/ihfs/architecture_doc/ihfs.htm), which provides definitions for and identifies interfaces between the software components which manipulate that data. Two other available pieces for the foundation are a proposed Logical Process Model and a proposed Software Development Process. With that foundation in place, the Hydrology Laboratory undertook an experiment to validate the components of that foundation by using them to develop a prototype application.

Within the overall goal of validating the foundation components, the project hoped to investigate or demonstrate a number of concepts. The primary intent was to show that an application could access data which currently resides in the legacy NWSRFS data system via a standards based, object oriented Application Program Interface (API) in keeping with the IHFS software architecture. Another important goal was to implement a portion of the previously developed IHFS Logical Data Model to assess its utility as a description of the data domain. The prototype would also be used to assess the feasibility of upgrading and evolving the NWSRFS via gradually replacing applications and portions of the system versus a single "big bang" approach.

The project team selected the MAPX preprocessor application as the basis for the prototype because while it accessed data from several different NWSRFS data stores, the application itself was relatively straightforward and simple to implement. The MAPX preprocessor computes mean areal precipitation values based upon gridded precipitation values derived from radar observations. These values are then stored back into an NWSRFS data store in a time series format.

II. Development Process

The proposed IHFS software development process was based heavily on an object-oriented approach using a Computer Aided Software Engineering (CASE) tool. The concept of the process was to create a design based around data objects from the LDM, use the CASE tool to generate code from the design, and then use the CASE tool to maintain the design and software as they evolved throughout integration and testing.

As the name suggests, CASE tools are intended to assist in the software engineering process. While different CASE tools provide assistance in various forms, the central idea is to eliminate the mechanical or tedious aspects of software design and development and allow the user to concentrate on ideas rather than process. For example, after an object is defined, the tool maintains that definition in a diagram and can automatically copy the object wherever the designer wants it in the design. The tool also ensures that definitions and interfaces are consistent throughout the entire design.

The LDM is defined using the "Unified Modeling Language" (UML). UML is recognized as "the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. It simplifies the complex process of software design, making a 'blueprint' for construction" (http://www.rational.com/uml/index.jsp). A design in UML manifests itself as a set of diagrams containing standard symbols that graphically depict the definitions and relationships of the components of the design.

The IHFS project uses the CASE tool, ObjectTeam, to create and maintain the LDM, so it was a logical choice for developing the prototype. (Note, this tool is currently distributed as "Telelogic Tau" by Telelogic AB). In addition to supporting object oriented design and analysis, ObjectTeam provides the capability to generate source code in a variety of languages from the UML diagrams.

The prototype team used the code generation feature to produce the initial source code for almost all of the software modules developed for the prototype. The source code generated by the tool essentially consists of a framework into which a developer still must insert code to actually implement most functions. The tool also provides a reverse engineering function which is somewhat able to analyze source code and update the class information held in the tool. While the function proved somewhat useful for adding and analyzing class libraries, it was not particularly useful for maintaining successive iterations of software modules during development. In fairness, this deficiency may have been due at least in part to a lack of investment of resources to thoroughly understand, customize and maintain the tool for our particular application.

The use of the ObjectTeam code generation function led to another even greater difficulty. The intent of the LDM was to be "language neutral", in other words, the model should be usable without regard to the language in which it was to be implemented. Unfortunately, the version of the tool which we used required different UML constructs depending upon whether c++ or java code was being generated. This deficiency may have been addressed in later versions of the tool, but we don't know because the prototype team was reluctant to change the tool during the project. As a result, the prototype currently maintains two slightly different sets of diagrams - one of which represents the c++ implementation and the other representing the java code.

During the prototype development effort, there were usually only two people actively involved in design and programming at any time. For efficiency, we attempted to divide the responsibility for detailed design and coding into sections with clearly defined interfaces. In view of the small team size and the fact that the objects and interfaces could be maintained in common by ObjectTeam, we expected that the amount of coordination and configuration management needed would be minimized. However, the team still found it necessary to periodically discuss the intent and details of the objects and interfaces during both the detailed design and the coding and integration phases of the project. It was also imperative to mandate that all development used a common set of object definitions.

III. The Prototype Design

As stated above, the first step in the development process is the design. In creating the design, the prototype team had to consider several important design constraints:

The prototype application had to replace its counterpart without any impact upon the regular process flow within NWSRFS. This constraint was complicated by the fact that most of the NWSRFS is written in Fortran and relies heavily upon common blocks to pass information; and the data stores have a custom format accessible through a large number of very specific routines also written in Fortran.

Another important design constraint was that all the objects used in the prototype had to be consistent with the IHFS Logical Data Model (LDM). Bonnin and Urban describe the logical data model as being "concerned with the data itself rather than how it [the data] is stored. It provides a complete and detailed description of the data, its attributes and interrelationships without regard for physical implementation." The design and implementation of the prototype led to several iterations of modifications and improvements to the LDM. The development team also found it necessary to compress layers of inheritance in order to reasonably efficiently implement the relationships between some of the objects used within the prototype. However, where this simplification was done, pains were taken to preserve the basic character of the relationships.

One design decision which evolved from the need for consistency with the LDM was that the objects upon which the prototype was built would be of two types: pure data (data-only) objects and process objects. While categorizing objects in this fashion seems contrary to most conventional object-oriented design theory, it was consistent with the software architecture and with the concept that the data layer would just be providing data through the data access API. In the actual implementation, for the most part, the data objects only contained methods to access (set and get) the data attributes they contained. The process objects primarily consisted of methods to manipulate external data. The latter objects are much like the concept of interfaces in java. Unfortunately, c++ doesn't provide a close counterpart to this concept.

The design of the MAPX application was constrained to be consistent with the "Logical Process Model."   The logical process model was not as mature and has not been subjected to the same level of internal review as the LDM. However, for the most part the only impact of this constraint was in the naming and relationships of the process objects.

The prototype development was to be implemented using commercially available or public domain software wherever possible, with the actual software written by the team being "as little as possible, but as much as necessary."

Some of the design constraints seemed easy to address initially. Since the prototype application was to be written in an object-oriented language, c++ was chosen as being the most widely accepted object-oriented language at the time.

However, the design team spent a lot of time trying to find the best way to accommodate some of the other constraints. One design constraint that was difficult to meet was that the application should access data at the object level. In other words, the application should accept and return data objects, rather than individual data elements or values. One of the most significant decision points involved the identification of the standard data interface to be used for the Application Program Interface (the interface between the Application layer and the DataAccess layer). The criteria used in this decision included: the breadth of recognition and acceptance of the standard, the stability of the standard, and the availability (and cost) of suitable COTS products which implemented the standard. The COTS product was a key element in the decision because it would form the heart of the DataAccess Layer. Essentially this layer provides the connection between the application and the data stores. For the prototype, the data stores are the legacy NWSRFS custom flat file system. However, the prototype also needed to demonstrate the feasibility of using the data access layer to isolate the application from the data store and allow the form of the data stores to evolve without requiring change to the applications. In accordance with the design constraints, the team looked for COTS packages that supported an object oriented API while interfacing with a flat file data structure.

After a great deal of research and discussion, the ODMG2 standard, defined and maintained by the Object Data Management Group (http://www.odmg.org) was selected as the most suitable interface standard for the API at that point in time (Summer, 1999).   Accompanying this decision was the decision to use the Visual Business Sight Framework (VBSF) from ObjectMatter, Inc. as a key component of the Data Access Layer. (Complete information about the tool is available from http://www.objectmatter.com .)

Two additional design constraints arose from the VBSF selection. First, VBSF required the use of the Java programming language. Second, VBSF assumed that the data was stored in a relational data base and accessible via the Java Data Base Connectivity (JDBC) protocol.   While these constraints introduced many additional complications for the design, the design team decided that the constraints were acceptable because of the potential long term benefits. Java was viewed as rapidly becoming a very widely accepted object oriented language and its use could provide additional flexibility in the choice of languages for future application development. Similarly, adoption of JDBC seemed like a reasonable choice because a long term goal was the transition of most, if not all of the NWSRFS data from flat files to a relational storage system. However, several of the complications are worth noting.

The introduction of Java into the middle layer of the prototype design meant that there would have to be at least two interfaces between different languages in the design. Java provides a relatively straightforward means of interfacing with c and c++. However, no good way was found to interface directly between java and Fortran and so another layer of c++ code was introduced as a bridge between java and Fortran.

The decision to use java in the prototype also required that most of the NWSRFS function libraries had to be remade as shared (position independent) rather than archive (or absolute) libraries. One of the limitations of the java language is that any external functions which it uses must be contained in shared libraries. Although the creation of the shared versions of the libraries was uneventful, thorough testing of all aspects of NWSRFS using shared libraries has not been conducted.

The need to support JDBC calls to access the data required the development of a driver which mapped those calls to the existing NWSRFS data access functions. As mentioned earlier, since the legacy data access functions were implemented in Fortran, the team had to develop wrappers which allowed the functions to be called from c++. In keeping with the minimalist approach to development, JDBC driver only supported the set of JDBC function calls which were actually used by VBSF to access the data used by the MAPX application. This same philosophy is embodied in the data access wrappers, which provide support only for those calls actually used to access data needed by the application.   However, in general the designs were intended to be easily extended to provide access to the remaining data in the NWSRFS data stores.

As the prototype team began to design and implement the DataAccessLayer interface between the application layer and the VBSF component, two discoveries were made. First, there were significant differences between the bindings, or specifications, of the ODMG2 standard in c++ and java. Secondly, the API that VBSF provided turned out to be “ODMG-based” rather than “ODMG-compliant.” In the ODMG2 standard, the c++ specification was substantially more mature and detailed than the java specification. This was probably a result of the fact that the java language itself was rapidly evolving and maturing in the same time frame as the ODMG2 standard. Basically, the result of these two factors combined with the language difference between the c++ of the application and the java of VBSF was that the interface became “much thicker” (i.e., more complex) and the prototype team had to develop the code to implement that interface. Furthermore, while the resulting interface would still be recognizable to anyone familiar with the ODMG2 standard, an independent application which was strictly compliant with the ODMG2 standard might require some adaptation to use the DataAccessLayer interface.

A graphic depiction of the thicker interface can be seen in the prototype diagram (Figure 1). The DataAccess API consists of all the objects between the two horizontal lines (dotted) labeled ODMG2. The VBSF interface is implemented in java in the Database object class. This very complex class includes a number of methods for creating, retrieving, updating deleting and otherwise manipulating objects and the underlying data of which the objects are composed. In order to make this interface callable from c++, the prototype implemented a corresponding object class in c++ named DataConnection which supported only the subset of methods which were actually needed to complete the MAPX application. Another class named JavaDataAPI was implemented in java to contain the java versions of the DataConnection methods and make the calls to the methods provided by the Database class.

As mentioned earlier, one of the more challenging design constraints was for the application to access data in terms of data objects rather than data elements. One of the consequences of the design to interface the object-oriented MAPX application, written in c++, with the object-oriented VBSF, written in java, is that each object which is passed across the language boundary is duplicated in the other language's memory space. The design components responsible for performing this duplication are the ObjectBuilders. The design includes a generic ObjectBuilder object and a specific ObjectBuilder for each of the main object classes which are passed across the DataAccess Interface. In the execution of the prototype, all of the data used by MAPX is initially retrieved from one of the NWSRFS data stores. Therefore, each data object is initially built in java by VBSF. When the data object passes across the language boundary to the DataConnection object, it invokes methods within the generic ObjectBuilder object and the specific object builders to create a duplicate data object in the c++ memory space. There are slight differences in the definitions of the data object attributes, due to differences within the languages themselves.

The development team believes that there may be significant room for improvement within the object building design. Currently, the objects are built according to "hard-coded" definitions of the class attributes and types. The development team identified the modification of the design to use a metadata definition that is available at execution as a logical enhancement to the design. VBSF uses the class definitions contained in the schema file to generate objects and it should be possible for the c++ builder objects to access this information. At the same time, it should be possible to streamline the code to take advantage of many similarities in the functions being performed.

Based on the testing that was done during development, the design of this interface seems to have a substantial impact on the overall performance of the prototype. During the retrieval of large objects such as a gridded precip file (with over 4,000 points), a significant amount of time is spent copying each grid point value from the java object to the c++ object. This issue should be closely reviewed for ways to improve the design to a point where it's performance would be operationally acceptable.

An early design decision led to a different path to handle the parameter data held in the Parameter Common Block (PCB). This data may be more widely known as the Hydrologic Control

Language (HCL) input. Since the parameter data used by MAPX is made available via a Fortran Common Block (in memory) rather than contained in a file, the design was simplified to allow the data access path to this data to bypass the VBSF middleware. Instead, the PCBConnection object directly invokes the PCBWrapper methods. These methods in turn wrap the individual Fortran data access functions used to read the HCL data. Since both PCBConnection and PCBWrapper are written in c++, the implementation is simplified, with no language translation needed. Modifying the design and implementation to allow this data to be accessed through the VBSF middleware should be straightforward, if that is desirable in the future. For example, if this data is to be stored in a data base.

The JDBC_IHFS_Driver is a java class library which emulates the JDBC compliant interface to a relational data base management system with which VBSF is designed to interface. Essentially, this means that the driver, receives a database command and acts upon it to perform the requested service or return the appropriate result set. In reality, the driver only responds to the subset of JDBC functions which VBSF uses to access the data needed for the MAPX application. However, since the functions are general purpose in nature, only a limited amount of enhancement should be needed to support the data requirements for all of the NWSRFS pre-processors. The design of the driver modules was based on the SimpleText model (Patel and Moss, 1997).

The JDBC_IHFS_Driver (in conjunction with the lower level components that it uses to communicate with the legacy data) provides a great deal of flexibility which could be useful in any future transitions or evolutions of the legacy data stores to a standard RDBMS. JDBC is a widely accepted standard interface between java applications and an RDBMS. Although the interface level is relatively low, we anticipate the availability of products which would map or translate between JDBC and a higher level standard such as XML.

The MiddleMan component actually spans the lower level boundary between c++ and java. While the module consists of both a java piece and a c++ piece, the java piece is strictly a calling point for the c++ piece which provides all the functionality. This component essentially does the translation between the data values in the columns of the emulated relational tables and the data tuples handled by the I/O wrappers.

The I/O wrappers are c++ functions which allow c++ routines to use the Fortran data access routines. The wrappers define data tuples which allow the data to handled in a standardized format by other c++ routines. They also provide a bridge between the very loose data type checking in Fortran and the strongly typed c++.

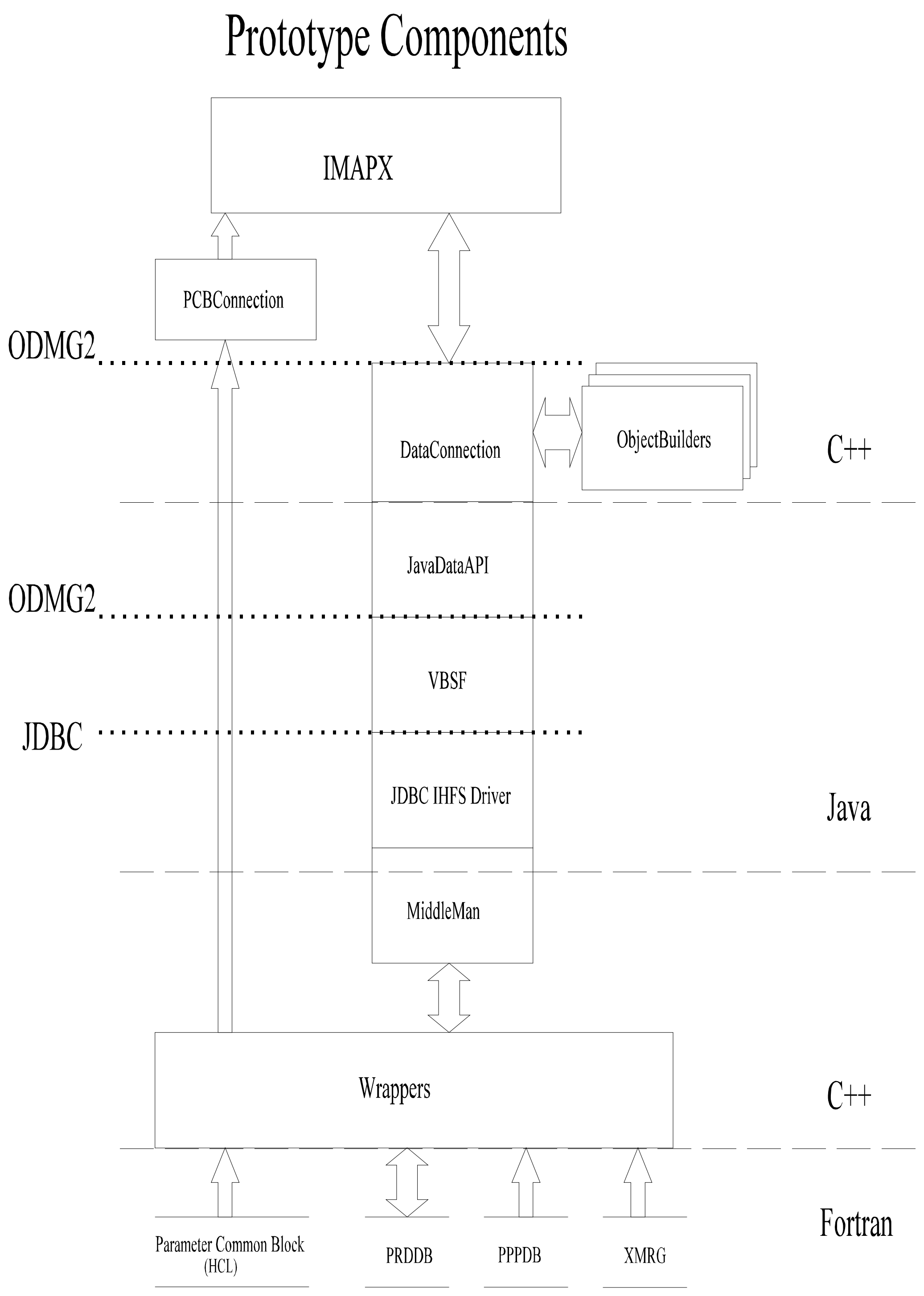
Prototype Components
IMAPX
PCBConnection
ODMG2
DataConnection
ObjectBuilders
C++
JavaDataAPI
ODMG2
VBSF
JDBC
JDBC IHFS Driver
Java
MiddleMan
Wrappers
C++
Parameter Common Block (HCL)
PRDDB
PPPDB
XMRG
Fortran

Figure 1.

IV. Observations from the Development Process

The issue of what standard should be used for the DataAccess API should probably be re-examined. At the time of the original decision there were many arguments in favor of the emerging XML standard, but it was not judged to be as mature or as widely accepted as ODMG2. While the ODMG standard is still widely accepted and has been refined in ODMG3, XML seems to be attracting more support and attention from developers and software vendors. Another point to consider is that as we worked with the ODMG2 standard, we identified some significant differences between the c++ and java bindings of the standard, this point may have been addressed in the ODMG3 standard, but should be considered before a decision is made to do further work using this standard.

Another lesson which seems to be relearned with every prototype development project is how easy it is to underestimate the effort required to do something which hasn't been done before. Two prime examples of this were the development of the c++ Data Access API and the JDBC IHFS Driver.

The development team expended considerable effort dealing with and overcoming the problems caused by working with the same objects in multiple programming languages. A particular problem is the differences in the data types supported or used by each language. This can be a problem even in cases where the differences seem minor or non-existent. For example, all three languages have the concept of an integer, but the definitions and properties of even integers are inconveniently different. Where it wasn't sufficient to use the lowest common denominator of the data types, the development team created a substitute in the necessary language. For example, a c++ implementation of a Number class which is the parent of all numeric types.

V. Observations about the Tools Used

A. ObjectTeam CASE tool

The ObjectTeam tool was very useful in the design phase of the prototype project. The UML diagrams were very helpful in conveying and maintaining the definition of objects and their relationships. The tool does basic error checking and balancing to ensure the existence of all methods and attributes that are referenced. The tool provides a project management hierarchy and version control scheme. This may be customizable, however, the prototype team simply adapted our approach to work within the standard tool definitions. The team also used the tool to create the initial source code for most of the objects which were developed and was generally pleased with the results and the resultant productivity gains over manually generating the initial code separately for each data class (module). This was especially true for the c++ code, where the .hxx files were often usable with minimal manual changes and captured many of the dependencies . In general for the .cxx and the .java files, the tool generated placeholders for the code to implement the methods defined for the class. This was also convenient and provided an easily identifiable indicator of methods which remained to be coded. On the negative side, for the "processing" classes, the tool often generate seemingly extraneous code and methods to accommodate the relationships between the processing classes. This may have been an issue that could have been solved through customization for our specific needs.

One goal for the CASE tool was to use it to maintain the design as the design and code evolved during development. The team found that it was possible, with sufficient care, to use the tool to analyze the manually updated code (that is, the source files which contained code added by the developers) and produce updated design diagrams. However, this process required that any manually added code had to be within delimited areas and the resultant diagrams had to be reviewed for accuracy. In addition, even if the diagrams accurately reflected the design of the latest code, the tool was not able to reliably regenerate the same source files from the diagrams. (Note that the tool's documentation specifically warned that this was not possible.) For these reasons, the tool did not really produce much benefit with respect to easily ensuring the synchronization of the code and the as-built designs.

B. VBSF

The VBSF tool consists of two parts: an off-line, mapping tool, which is used to define the relationships and mappings of the objects to the relational tables in which the data is stored; and a real-time java class library which provides the functions which actually provide the bridge between objects and relational tables during execution.

The VBSF tool seemed to do a good job at mapping objects to relational tables. There are a number of subtleties involved in mapping the relationships and even after some extensive experimentation, the prototype development team is not convinced that they have achieved the optimum mapping. The software support via email for this product is very responsive and during the prototype project, several upgrades to the product were announced. Unfortunately, the prototype was unable to take advantage of these because the software required a later version of java which was not supported by the target platform operating system (HPUX 10.2). In several cases, the prototype developers had to design and implement workarounds for deficiencies which were addressed in later VBSF versions. This tool should definitely be considered if there are future needs to interface java code with data stored in relational tables

VI. Conclusions

The IHFS Data Access Prototype project has demonstrated that the Logical Data Model is a valid portrayal the hydrologic data domain and therefore, should be useful as a guideline for evolving the current physical implementation of the IHFS database. The prototype also has shown that it is possible to hide the proprietary nature of the legacy NWSRFS data file structure beneath a standard data interface. These facts may allow a smoother transition and evolution of the

NWSRFS in the future. The project has allowed the team and therefore the Hydrology Lab to gain experience with the benefits and weaknesses of software development using CASE tools and with an object-oriented approach. The performance of the prototype calls into question the feasibility of implementing object-oriented interfaces in multiple languages at least using the current prototype design.

VII. References

Bonnin, G., and D. Urban, “Development Of A Data Architecture For The NWS Hydrologic Services Program,” *16th Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrolog*y, 2000.

Bonnin, G., J. Gofus, Y. Qu, and D. Urban, “Prototyping the Data Access Layer of the Integrated Hydrologic Forecast System,” *4th International Conference on Hydroinformatics,* July 2000.

Cattell, R.G.G., and D. Barry, editors, <u>The Object Database Standard: ODMG 2.0</u>, Morgan Kaufmann Publishers, 1997.

Patel, P. and K. Moss, <u>Java Database Programming with JDBC</u>, 2nd. Edition, Coriolis Group Books, 1997.